

DE MYSTERIIS DOM JOBSIVS: MAC EFI ROOTKITS

SNARE
@ SYSCAN SINGAPORE
APRIL 2012



assurance

AGENDA

► Things I will talk about

- I. Introduction - goals, concepts & prior work
- II. EFI fundamentals
- III. Doing bad things with EFI
- IV. Persistence
- V. Defence against the dark arts



I. INTRODUCTION



INTRODUCTION

I WANT A COOL BOOT SCREEN ON MY MAC

- ▶ Why are we here?
 - ▶ I wanted to mess with pre-boot graphics (seriously)
 - ▶ Minimal knowledge of firmware / bootloader
 - ▶ Did some research...
 - ▶ Wait a minute, backdooring firmware would be badass
 - ▶ But, of course, it's been done before...



INTRODUCTION

PRIOR ART

- ▶ Other work in this area
 - ▶ Old MBR viruses
 - ▶ ...
 - ▶ John Heasman @ Black Hat '07 (badass talk on EFI)
 - ▶ Core Security @ CanSecWest '09 (BIOS infection)
 - ▶ Invisible Things @ Black Hat '09 (Intel BIOS [UEFI])
 - ▶ and more...
 - ▶ see also endrazine's talk at Hackito Ergo Sum this year



INTRODUCTION

ROOTKIT/BOOTKIT/RED FISH/BLUE FISH

▶ Rootkit?

- ▶ Provide persistent access to an owned machine
- ▶ Historically, two main kinds
 - ▶ Kernel land - kernel module/etc
 - ▶ User land - patched binaries/LD_PRELOAD/etc
- ▶ These days - more low level badassery
 - ▶ Bootloader (“bootkit”)
 - ▶ Firmware/BIOS
 - ▶ SMM
 - ▶ ACPI
 - ▶ ...



INTRODUCTION

GOALS

- ▶ Backdoor a machine
 - ▶ Without evidence on-disk
 - ▶ Persist forever!
 - ▶ Across reboots, reinstalls, disk replacement, heat death of the universe
 - ▶ Patch the kernel at boot time
 - ▶ Work regardless of whole-disk encryption
- ▶ Sound hard?
 - ▶ Nah
 - ▶ (OK yeah, kinda - this is very much ongoing research)



I. EFI FUNDAMENTALS



WHAT'S AN EFI?

AND WHY DO I CARE?

▶ BIOS replacement

- ▶ Initially developed at Intel
- ▶ Designed to overcome limitations of PC BIOS
- ▶ “Intel Boot Initiative”
- ▶ Used in all Intel Macs - now I care

▶ UEFI

- ▶ Handed over to Unified EFI Consortium @ v1.10
- ▶ Became UEFI for v2.0+
- ▶ Apple's version reports as v1.10
- ▶ Used on lots of PC mobos



EFI ARCHITECTURE

PUTTING THE “FUN” IN “FUNDAMENTALS”!

► Modular

- Comprises core components, apps, drivers, bootloaders
- Core components reside on firmware
 - Along with some drivers
- Applications & 3rd party drivers
 - Reside on disk
 - Or on firmware data flash
 - Or on option ROMs on PCI devices



EFI ARCHITECTURE

TERMINOLOGY

▶ Protocols

- ▶ Chunks of firmware/driver functionality
- ▶ e.g. SimpleTextInput - console input

▶ Device Handles

- ▶ Groups of protocols per device

▶ CSM

- ▶ Compatibility Support Module
- ▶ PC BIOS emulation
- ▶ e.g. BootCamp



EFI ARCHITECTURE

TERMINOLOGY

▶ GPT

- ▶ GUID Partition Table
- ▶ Part of the EFI spec
- ▶ Required for booting from a disk

▶ ESP

- ▶ EFI System Partition
- ▶ 200MB FAT partition at beginning of GPT
- ▶ Apple only uses it for firmware updates
- ▶ Can store drivers here

▶ Everything has a GUID



EFI ARCHITECTURE

TERMINOLOGY

- ▶ Tables - pointers to functions & EFI data
 - ▶ System table
 - ▶ Pointers to core functions & other tables
 - ▶ Boot services table
 - ▶ Functions available during EFI environment - useful!
 - ▶ Memory allocation
 - ▶ Registering for timers and callbacks
 - ▶ Installing/managing protocols
 - ▶ Loading other executable images



EFI ARCHITECTURE

TERMINOLOGY

- ▶ Tables - pointers to functions & EFI data
 - ▶ Runtime services table
 - ▶ Functions available during pre-boot & while OS is running
 - ▶ Time services
 - ▶ Virtual memory - converting addresses from physical
 - ▶ Resetting system
 - ▶ Capsule management
 - ▶ Variables (we will use this)
 - ▶ NVRAM on the Mac - boot device is stored here
 - ▶ Configuration table
 - ▶ Pointers to data structures for access from OS
 - ▶ Custom runtime services



EFI ARCHITECTURE

DEVELOPING FOR EFI

- ▶ EDK2 - EFI Development Kit
 - ▶ Includes “TianoCore” - Intel’s reference implementation
 - ▶ Most of what Apple uses
 - ▶ And probably most other IBVs
 - ▶ Written in C
 - ▶ Builds PE executables
 - ▶ Main types of executables
 - ▶ Core components - SEC, PEIM, DXE, BDS
 - ▶ Applications - e.g. EDK shell (see rEFIt)
 - ▶ Drivers - support hardware
 - ▶ Bootloaders



EFI ARCHITECTURE

STATS

- ▶ EDK2 has >2million lines in .c/.h files
 - ▶ Compared to ~1.1mil in XNU
 - ▶ ~14mil in Linux
 - ▶ `find . \( -name "*.c" -o -name "*.h" \) |xargs cat|wc -l`
 - ▶ (not very scientific, whatever)
- ▶ Spec is 2156 pages long at v2.3.1



EFI ARCHITECTURE

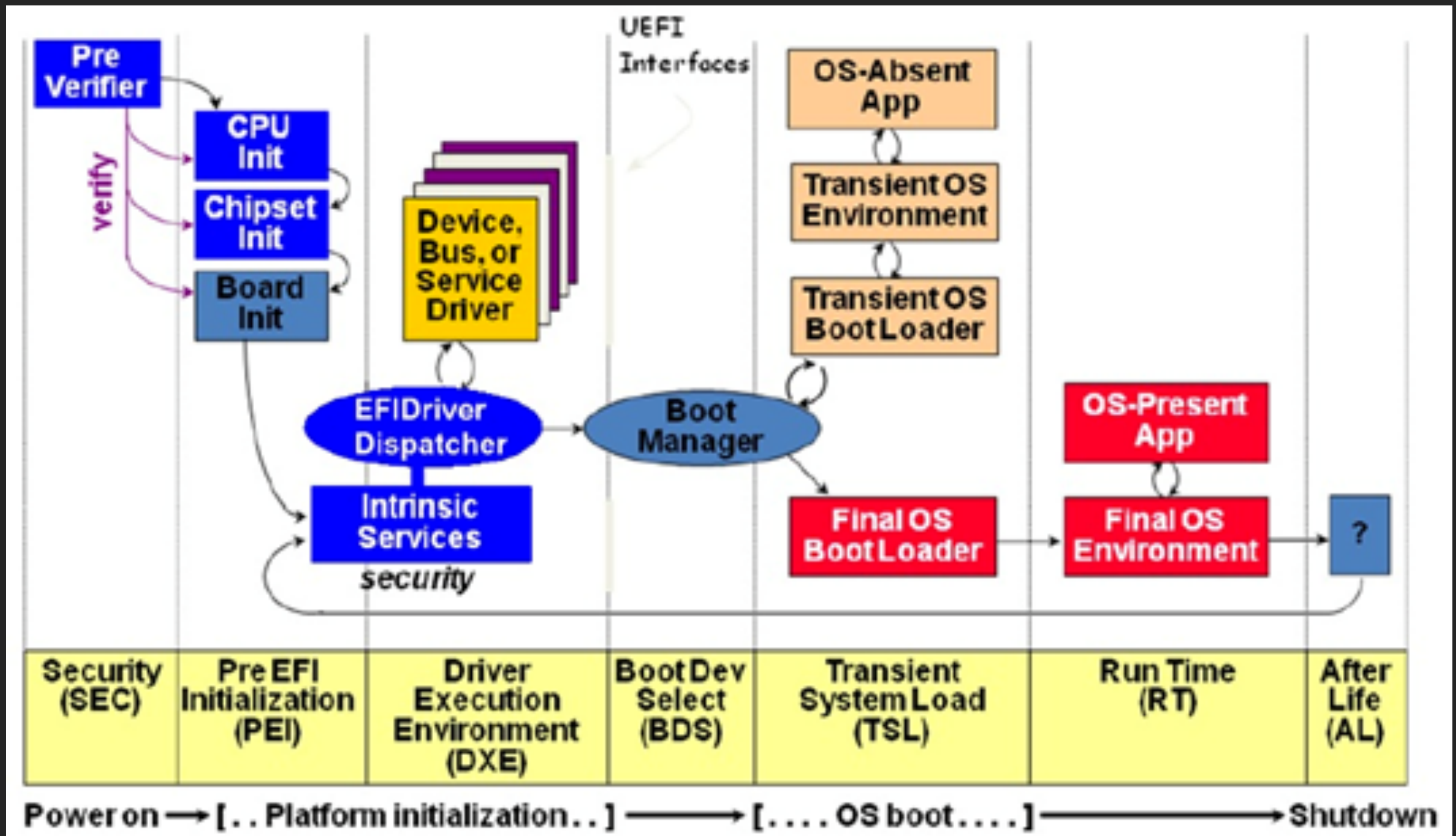
STATS

- ▶ Some telling examples of defined protocols
 - ▶ Disk/filesystem access, console input/output
 - ▶ Graphics Output Protocol, Human Interface Infrastr.
 - ▶ IPv4, IPv6, TCP, UDP, IPSEC, ARP, DHCP, FTP, TFTP
 - ▶ User management, SHA crypto, key management
- ▶ Starting to sound like an entire OS



EFI ARCHITECTURE

BOOT PROCESS



Token shitty, low res diagram stolen from documentation



II. DOING BAD THINGS WITH EFI



DOING BAD THINGS WITH EFI

WHAT CAN WE DO?

- ▶ Modularity makes it pretty easy
 - ▶ Build a rogue driver
 - ▶ Get loaded early on
 - ▶ Register callbacks
 - ▶ Hook Boot Services/Runtime Services
 - ▶ Hook various protocols
- ▶ No awful 16-bit real-mode assembly necessary
- ▶ Generic interface - minimal platform-specific stuff



DOING BAD THINGS WITH EFI

ATTACKING WHOLE-DISK ENCRYPTION

- ▶ How does “FileVault 2.0” work?
 - ▶ Disk has ESP, encrypted OS partition, “recovery” partition
 - ▶ Platform firmware inits
 - ▶ Loads bootloader from “recovery” partition
 - ▶ Bootloader prompts user for passphrase
 - ▶ Uses passphrase to decrypt AES key off disk
 - ▶ Uses AES key to decrypt kernel(cache)
 - ▶ Execute kernel



DOING BAD THINGS WITH EFI

ATTACKING WHOLE-DISK ENCRYPTION

- ▶ Stealing the user's passphrase
 - ▶ Keystroke logger!
 - ▶ Hook the SimpleTextInput protocol
 - ▶ Specifically, the instance installed by the bootloader
 - ▶ Replace pointer to ReadKeyStroke() with our function
 - ▶ Every time a key is pressed, we get called
 - ▶ Record keystroke, call real ReadKeyStroke()



DOING BAD THINGS WITH EFI

ATTACKING WHOLE-DISK ENCRYPTION

- ▶ Steal the AES key
 - ▶ I haven't actually tried this
 - ▶ Hook LoadImage () function in Boot Services
 - ▶ Patch the bootloader when it is loaded
 - ▶ Shouldn't be toooooo hard...

```
aStartUnlockcor db 'Start UnlockCoreStorageVolumeKey',0
                  ; DATA XREF: start+481↑o
                  align 8
aEndUnlockcores db 'End UnlockCoreStorageVolumeKey',0
                  ; DATA XREF: start+49F↑o
                  align 8
```

(thanks for the debug logging, Apple)



THEY'RE GOING AFTER THE KERNEL!



**OTTERZ?
IN MY
KERNEL?**



ATTACKING THE KERNEL

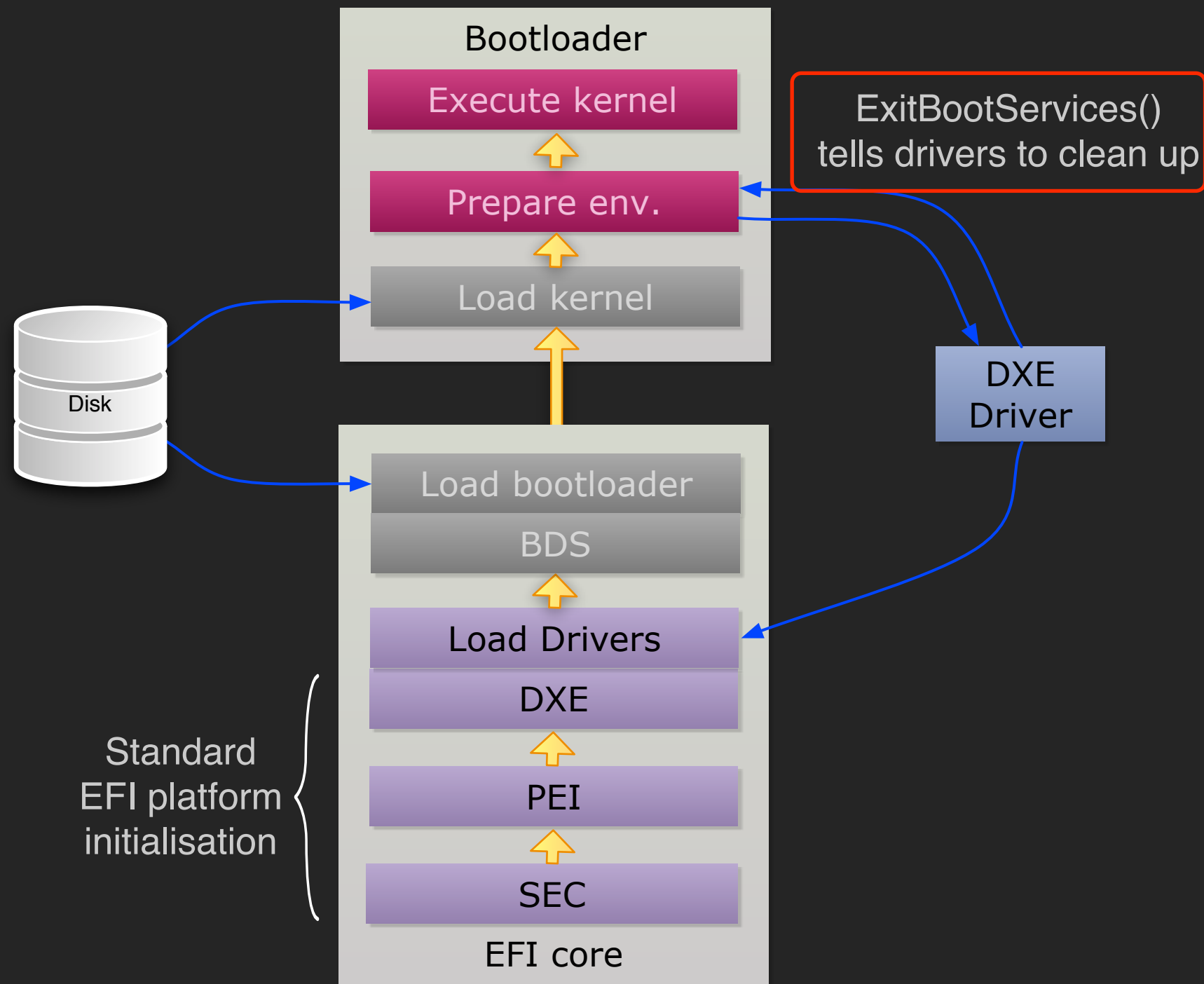
WHAT CAN WE DO?

- ▶ Patch the kernel from EFI
 - ▶ Find some place to put code
 - ▶ Hook some kernel functionality
 - ▶ Get execution during kernel init
 - ▶ Party
- ▶ It's not loaded when we get loaded
 - ▶ So how do we trojan the kernel?
 - ▶ Wait until it is loaded, then POUNCE
 - ▶ `ExitBootServices()`



ATTACKING THE KERNEL

EFI BOOT PROCESS



ATTACKING THE KERNEL

WHERE IS IT?

Start of kernel image is at 0xffffffff8000200000

```
$ otool -l /mach_kernel
```

```
/mach_kernel:
```

```
Load command 0
```

```
    cmd LC_SEGMENT_64
```

```
cmdsize 472
```

```
segname __TEXT
```

```
vmaddr 0xffffffff8000200000
```

```
vmsize 0x00000000000052e000
```

First kernel segment VM load addr



```
gdb$ x/x 0xffffffff8000200000
```

```
0xffffffff8000200000: 0xfedfacf
```



Mach-O header magic number (64-bit)



ATTACKING THE KERNEL

PATCHING THE KERNEL

- ▶ We know the kernel is at `0xffffffff800020000`
 - ▶ EFI uses a flat 32-bit memory model without paging
 - ▶ In 32-bit mode its at `0x00200000`
- ▶ What do we do?
 - ▶ Inject a payload somewhere
 - ▶ Patch a kernel function and point it at the payload
 - ▶ Trampoline payload to load bigger second stage?
 - ▶ From an EFI variable
 - ▶ From previously-allocated Runtime Services memory
 - ▶ Over the network



ATTACKING THE KERNEL

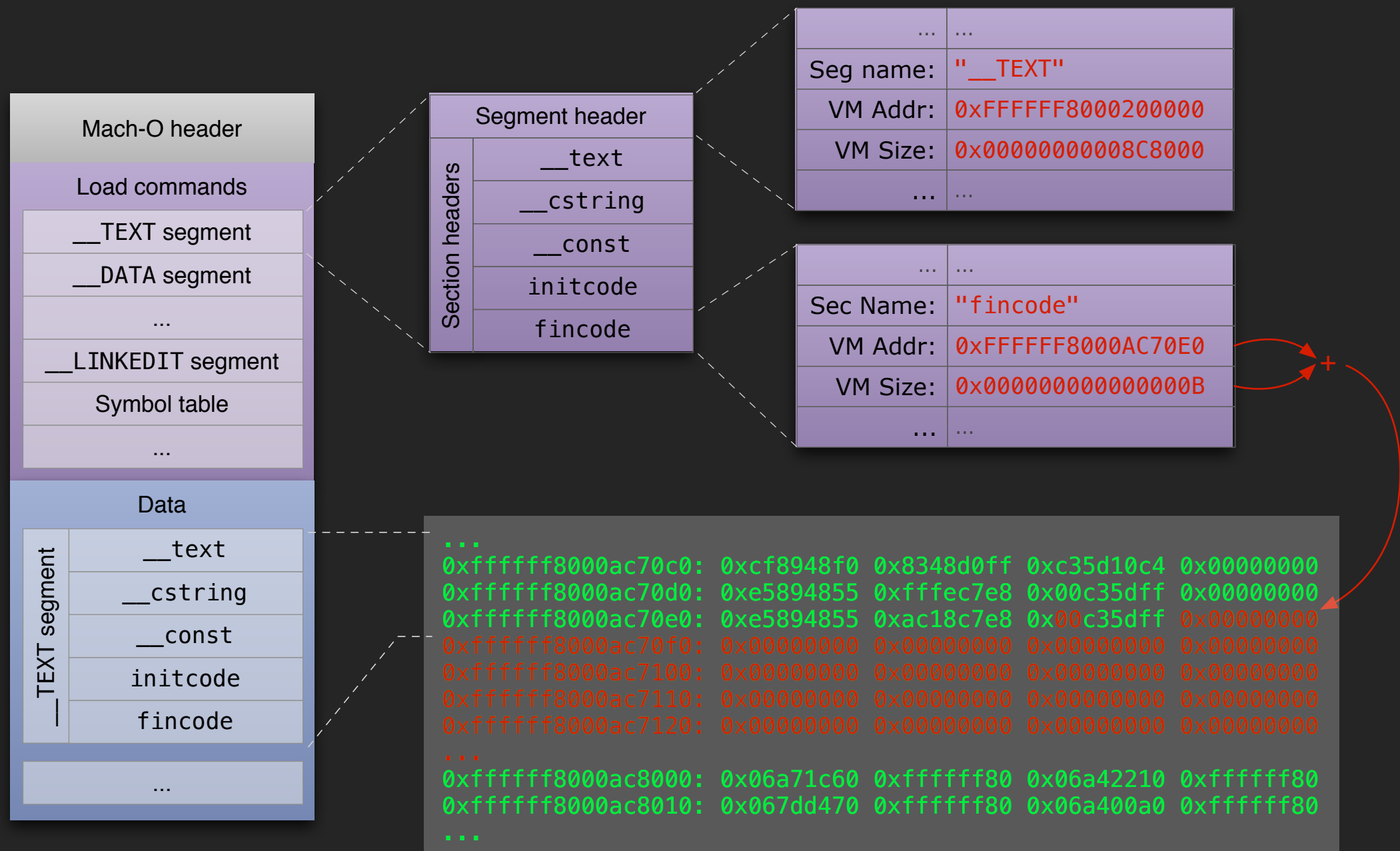
PATCHING THE KERNEL

- ▶ Where can we put our payload?
 - ▶ Page-alignment padding
 - ▶ End of the `__TEXT` segment
 - ▶ On the default 10.7.3 kernel, almost an entire 4k page
 - ▶ WIN



ATTACKING THE KERNEL

PATCHING THE KERNEL



ATTACKING THE KERNEL

PATCHING THE KERNEL

▶ OK, so

- ▶ We have been called by `ExitBootServices()`
- ▶ We know where we can store a payload
 - ▶ And how much space we have
- ▶ What do we put there?
- ▶ And how do we get it called?



ATTACKING THE KERNEL

PATCHING THE KERNEL

► What's our payload? Tramampoline!

- Save registers
- Locate next stage payload
 - Stored in an EFI variable
- Call next stage initialisation
- Restore patched instruction
- Restore registers
- Jump back to patched func
- Kernel continues booting



ATTACKING THE KERNEL

PATCHING THE KERNEL

- ▶ How do we get it called?
 - ▶ We patch a function in the kernel's boot process
 - ▶ `load_init_program()` is a good candidate
 - ▶ Kernel subsystems are mostly initialised
 - ▶ We're ready to exec the init process
 - ▶ Save the first instruction in the function, store in payload
 - ▶ Overwrite it with a jump to our payload



ATTACKING THE KERNEL

PATCHING THE KERNEL

► Preparing our trampoline

```
/* We're going to patch the first instruction of load_init_program(), and
 * we need to jump back here */
tramp.patch_addr = find_kernel_symbol("_load_init_program");
DLOG(L"[+] patching load_init_program @ 0x%p\n", tramp.patch_addr);

/* Save the instruction data that we're going to overwrite. The tramp will
 * fix it up afterwards. */
tramp.patch_save = *((uint64_t *)tramp.patch_addr);
DLOG(L"[+] saved instructions: 0x%llx, \n", tramp.patch_save);

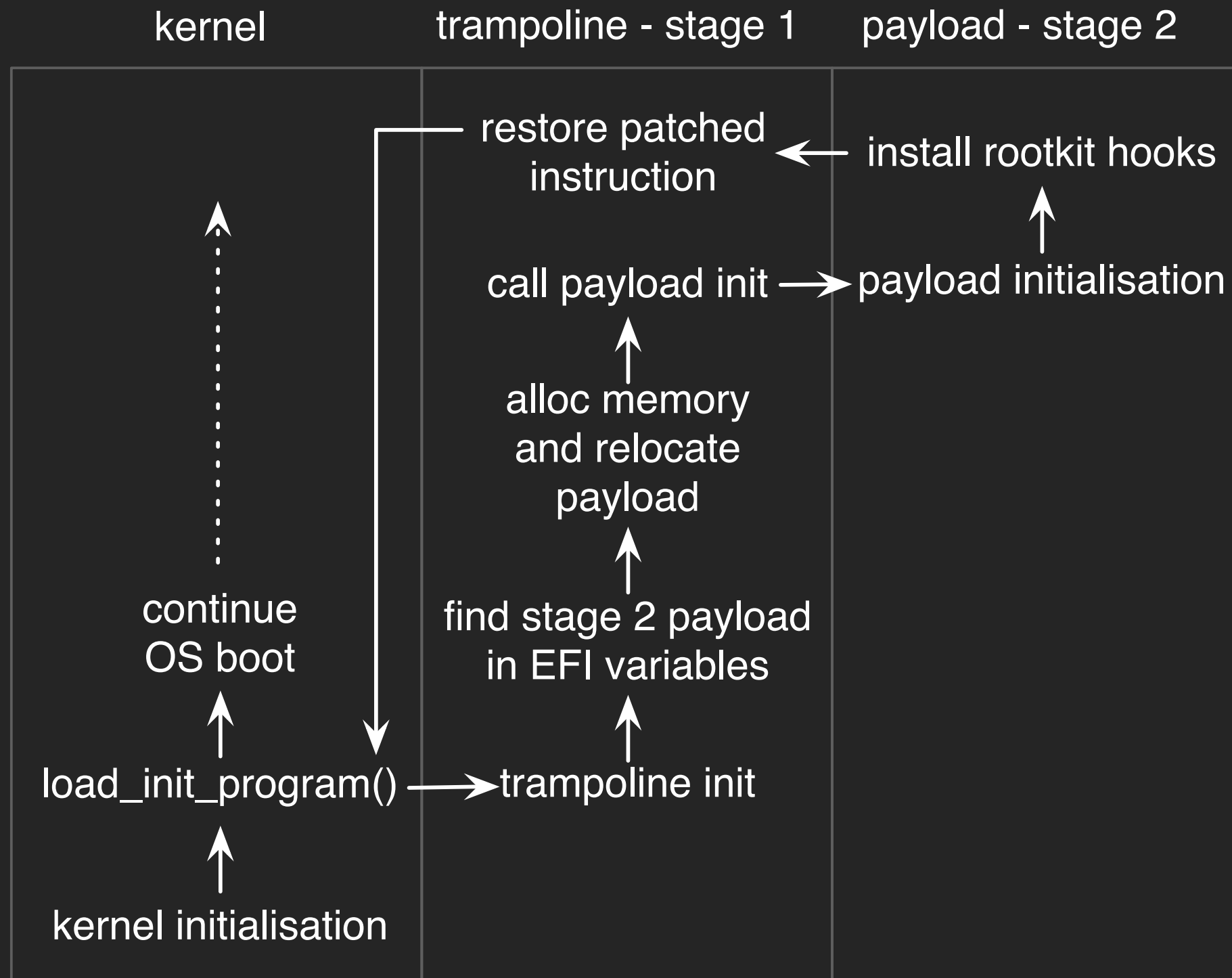
/* Overwrite the instruction with a jump to the trampoline shellcode */
jump.displacement = (uint32_t)sc_start - (uint32_t)tramp.patch_addr -
                    sizeof(jump);
*((uint64_t *)tramp.patch_addr) = *((uint64_t *)&jump);
DLOG(L"[+] patched with instruction: 0x%llx\n", *((uint64_t *)&jump);
```

► Then we just copy it into the kernel



ATTACKING THE KERNEL

PATCHING THE KERNEL



ATTACKING THE KERNEL

HALF-ASSED ROOTKIT HOOKS SLIDE

- ▶ What do we do once we're in the kernel?
 - ▶ Minimal detail here...
 - ▶ See my blog for previous talks on XNU rootkits, etc (<http://ho.ax>)
 - ▶ See fG's blog for more rad stuff (<http://reverse.put.as>)
 - ▶ Hook syscalls
 - ▶ Install NKE callbacks (socket/IP/interface filters)
 - ▶ Install TrustedBSD policy handlers
 - ▶ Patch things
 - ▶ ... and so on



ATTACKING THE KERNEL

OTHER HALF-ASSED ROOTKIT HOOKS SLIDE

- ▶ e.g. Hooking the `kill()` syscall
 - ▶ Demo will use this
 - ▶ Overwrite entry in `sysent` to point to our function
 - ▶ Our function...
 - ▶ Checks for a special condition (signal number == 7777)
 - ▶ Promotes the calling process to uid 0
 - ▶ Calls the original `kill()`

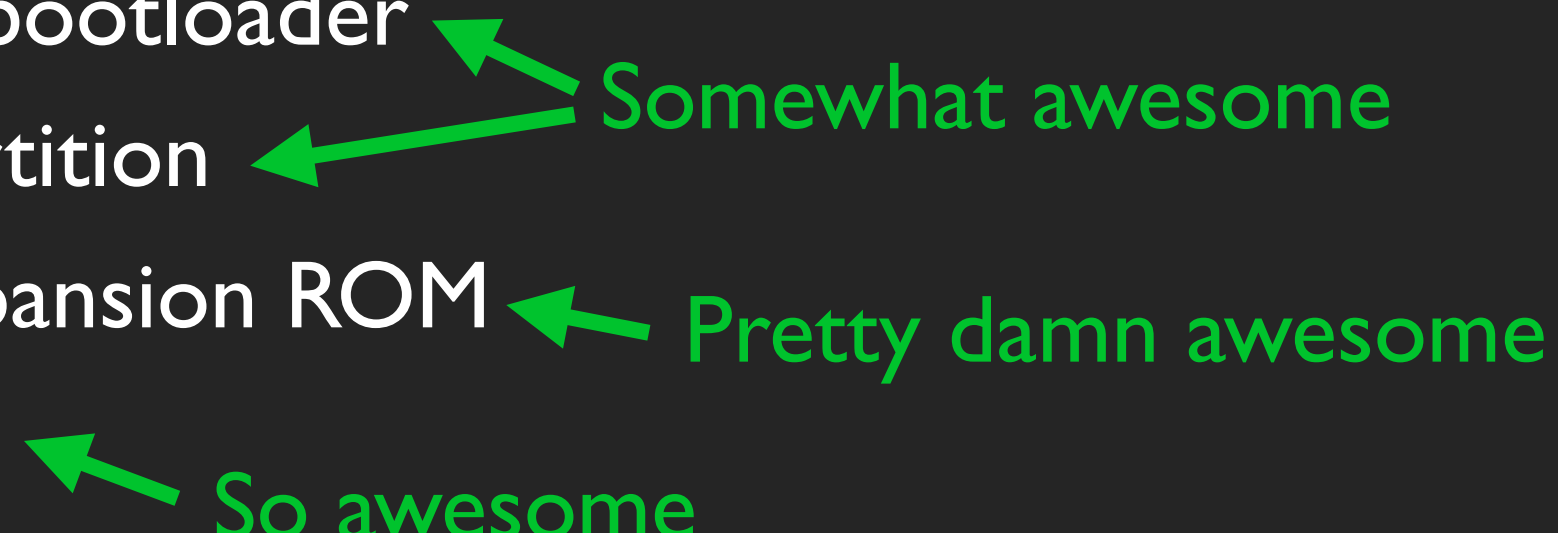


IV. PERSISTENCE



PERSISTENCE

OPTIONS?

- ▶ In ascending order of awesome
 - ▶ Patch/replace bootloader
 - ▶ EFI System Partition
 - ▶ PCI device expansion ROM
 - ▶ Firmware flash
- Somewhat awesome
- Pretty damn awesome
- So awesome
- 



PERSISTENCE

MESSING WITH THE BOOTLOADER

- ▶ `/System/Library/CoreServices/boot.efi`
- ▶ On-disk, why not just...
 - ▶ Patch the kernel
 - ▶ Install a kernel extension
- ▶ Somewhat useful for “evil maid” attacks
 - ▶ Even with FileVault, boot.efi is stored unencrypted
- ▶ Meh. 4/10.



PERSISTENCE

EFI SYSTEM PARTITION

- ▶ Not used by Apple's firmware
 - ▶ Except for firmware updates
- ▶ AFAIK no drivers are loaded automagically from here
- ▶ In this case - SUX. 0/10.



PERSISTENCE

PCI DEVICE EXPANSION ROMS

- ▶ Hardware-specific
- ▶ Loaded during DXE phase
- ▶ Graphics cards in iMacs have them
 - ▶ Probably MacBook Pros too
 - ▶ My old test MacBook - no dice
 - ▶ VMware's ethernet interfaces do - hurr (good for testing)
- ▶ Can write to them from the OS
 - ▶ Thanks, iMacGraphicsFWUpdate.pkg!
- ▶ Pretty awesome. **7/10.**



PERSISTENCE

FIRMWARE FLASH

- ▶ Hardware-specific, but it's always there
- ▶ Can modify everything
 - ▶ SEC, PEI, DXE, BDS, custom drivers, whatever
- ▶ Can be written to from the OS
- ▶ So awesome. **11/10** A+++++ would buy again.



PERSISTENCE

FIRMWARE FLASH

- ▶ Apple's firmware updates
 - ▶ Firmware updates are copied to ESP
 - ▶ Written to flash on reboot
 - ▶ Older machines use EFI Firmware Volumes (.fd files)
 - ▶ Volume is blessed with EfiUpdaterApp.efi
 - ▶ Writes to flash via SPI from EFI environment
 - ▶ Newer machines use EFI Capsules (.scap files)
 - ▶ EFI capsule mailbox stuff? (see the spec)
 - ▶ We can do it from a running OS with flashrom :)



PERSISTENCE

FIRMWARE FLASH

► Manipulating firmware data

- Both capsules and firmware volumes are in the spec
 - <http://download.intel.com/technology/framework/docs/Capsule.pdf>
 - <http://download.intel.com/technology/framework/docs/Fv.pdf>
- A capsule has a firmware volume inside
- Inside the FV is a set of Firmware Filesystem “files”
 - <http://download.intel.com/technology/framework/docs/Ffs.pdf>
- There are tools for manipulating Phoenix/AMI/etc BIOSes
 - Aimed at SLIC mods etc
- I wrote my own in python
- PS. Binaries are PE, remember? IDA understands them.



PERSISTENCE

FIRMWARE FLASH

[Firmware Volume]

Offset = 0x0 (0)

FileSystemGuid = 7a9354d9-0468-444a-81ce-0bf617d890df

FvLength = 0x190000 (1638400)

Signature = '_FVH'

Attributes = 0xffff8eff

HeaderLength = 0x48 (72)

Checksum = 0xdefd (57085)

Revision = 0x1 (1)

[FvBlockMap]

NumBlocks 25, BlockLength 65536

Files:

11527125-78b2-4d3e-a0df-41e75c221f5a (EFI_FV_FILETYPE_PEIM)

4d37da42-3a0c-4eda-b9eb-bc0e1db4713b (EFI_FV_FILETYPE_PEIM)

35b898ca-b6a9-49ce-8c72-904735cc49b7 (EFI_FV_FILETYPE_DXE_CORE)

c3e36d09-8294-4b97-a857-d5288fe33e28 (EFI_FV_FILETYPE_FREEFORM)

bae7599f-3c6b-43b7-bdf0-9ce07aa91aa6 (EFI_FV_FILETYPE_DRIVER)

b601f8c4-43b7-4784-95b1-f4226cb40cee (EFI_FV_FILETYPE_DRIVER)

51c9f40c-5243-4473-b265-b3c8ffa9fa (EFI_FV_FILETYPE_DRIVER)

... snip ...



DEMO TIME

I MADE OFFERINGS TO THE DEMO GODS

- ▶ Simple PoC rootkit - “defile”
- ▶ Loading driver from USB flash disk
 - ▶ Plug in flash drive with custom loader
 - ▶ Load malicious driver
 - ▶ Driver registers callback, hits `ExitBootServices()`
 - ▶ Store main payload in EFI variable
 - ▶ Patch kernel with trampoline
 - ▶ Tramp grabs rootkit payload from EFI var and installs it
 - ▶ Payload hooks `kill()` syscall
 - ▶ Everyone gets laid



DEMO TIME

I MADE OFFERINGS TO THE DEMO GODS

- ▶ Trojaned option ROM
 - ▶ Load malicious driver from option ROM
 - ▶ ... same as the other one



DEMO



V. DEFENCE



DEFENCE

ATTACK VECTORS - HOW YOU GOT OWNED

Remote exploit



Local privesc



Bootloader patch

Firmware flash

Option ROM flash



Network boot

Physical access



"Evil Maid"



DEFENCE

EFI FIRMWARE PASSWORD?

- ▶ Hahaha... :(
- ▶ This will prevent some “evil maid” attacks
- ▶ Stops you from changing the boot target
 - ▶ USB/Optical/Firewire/Network
- ▶ That's about it
- ▶ Doesn't prevent flashing the firmware from the OS
 - ▶ Or option ROMs
- ▶ There are ways to remove it



DEFENCE

UEFI SECURE BOOT

- ▶ Part of the current UEFI spec
- ▶ Describes signing of EFI images (drivers/apps/loaders)
 - ▶ Platform Key (PK)
 - ▶ Establish trust between platform owner and firmware
 - ▶ Enrol KEKs (sign KEK database)
 - ▶ Device may be distributed in setup mode (no PK) or user mode
 - ▶ Key Exchange Key (KEK)
 - ▶ Establish trust between the OS and the firmware
 - ▶ Sign image signature databases
 - ▶ Whitelist/blacklist of allowed/forbidden images
- ▶ DXE & BDS phases verify sigs of binaries



DEFENCE

UEFI SECURE BOOT

► Issues

- *“The public key must be stored in non-volatile storage which is tamper and delete resistant.”*
 - May not prevent evil maid attacks if NVRAM can be reset
 - Blank NVRAM == back to “setup” mode
- OS has KEK to enrol images in sig databases
 - Malware access to ring 0 == access to keys to enrol whatever
- Signing needs to be enforced through whole stack
 - If unsigned kernel drivers can be loaded, see previous point



IN CONCLUSION...

I HAD FUN.

- ▶ So basically we're all screwed
 - ▶ What should you do?
 - ▶ Glue all your ports shut
 - ▶ Use an EFI password to prevent basic local attacks
 - ▶ Stop using computers, go back to the abacus
 - ▶ What should Apple do?
 - ▶ Implement UEFI Secure Boot (actually use the TPM)
 - ▶ Use the write-enable pin on the firmware data flash properly
 - ▶ NB: They may do this on newer machines, just not my test one
 - ▶ Audit the damn EFI code (see Heasman/ITL)
 - ▶ Sacrifice more virgins



REFERENCES

- ▶ UEFI Spec
 - ▶ <http://www.uefi.org/specs/>
- ▶ EFI Development Kit II source & documentation
 - ▶ <http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2>
- ▶ Mac OS X Kernel Programming Guide
 - ▶ <http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/>
- ▶ Mac OS X Internals - Amit Singh
 - ▶ <http://osxbook.com/>
- ▶ Mac OS X Wars: A XNU Hope - nemo
 - ▶ <http://www.phrack.com/issues.html?issue=64&id=11#article>
- ▶ Runtime Kernel kmem Patching - Silvio Cesare
 - ▶ <http://biblio.l0t3k.net/kernel/en/runtime-kernel-kmem-patching.txt>
- ▶ Designing BSD Rootkits - Joseph Kong
 - ▶ <http://nostarch.com/rootkits.htm>
- ▶ Reverse Engineering Mac OS X - fG
 - ▶ <http://reverse.put.as/>
- ▶ Hacking The Extensible Firmware Interface - John Heasman
 - ▶ <https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf>
- ▶ Attacking Intel BIOS - Invisible Things Lab
 - ▶ <http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf>



KTHXBAI \m/

twitter: @snare

blog: <http://ho.ax>

greetz:

y0l l, wily, fG!, metlstorm, tmasky, andrewg

special thanks to:

thomas & co for a great conference

jesse for the test machine that I bricked (R.I.P. for now)

baker for the brutal art



assurance